Fortran coarray library for 3D cellular automata microstructure simulation

Anton Shterenlikht

Mech Eng Dept, University of Bristol, University Walk, Bristol BS8 1TR, UK
mexas@bris.ac.uk

Abstract

Coarrays are a part of Fortran language standard. In this work coarrays are used to construct a parallel cellular automata Fortran library for microstructure simulation. The library uses a range of multi-dimensional real, integer and logical coarrays with a three-dimensional image grid. A range of coarray synchronisation and collective statements are used. The performance of the library is measured using several test programs on HECToR, including solidification and fracture simulations. Models in excess of 10^{10} cells have been successfully analysed on up to $2^{15} = 32768$ cores with a speed-up of 20 compared to runs on 512 cores. While the library delivers useful microstructural predictions, the scaling results are disappointing, indicating the need for code optimisation. The library is freely available, under BSD license, from: http://eis.bris.ac.uk/∼mexas/cgpack.

1 Cellular Automata for Microstructure Simulation

Cellular automata (CA) is a popular microstructure simulation tool. It is a discrete space – discrete time method, in which space is divided into many small identical cells with a number of pre-defined states. State of each cell changes based on the states of some neighbouring cells. A CA model is often coupled with a finite element model to result in a hybrid discrete/continuum, multi-scale mechanical model. Many such hybrid models have been successfully used for the prediction of the ductile to brittle fracture [1], oxide cracking in hot rolling [2], grain instability [3], solidification [4] and recrystallisation [5]. Previously we have created a serial CA micro- and nano-structure evolution library, implemented in Fortran 2003 [3]. In this paper we present an attempt to parallelise this library with Fortran 2008 coarrays.

2 Fortran Coarrays

Coarrays are a part of the Fortran 2008 standard [6]. Coarray collectives are at a draft technical specification stage [7] and are expected to be part of the next minor revision of the standard. However, Cray Fortran compiler provides coarray collectives as an extension to the standard, with syntax and function very close to the technical specification. At this time coarrays are fully supported only by Cray and Intel compilers. G95 and GCC compilers provide partial coarray support (e.g. GCC supports only a single image) [8].

Coarrays are intended for SIMD type parallel programs. Multiple copies of the executable (*images*) are distributed to processors. If an array is declared as a coarray then a copy of this array is created on every processor. Each processor has full read/write access to coarray elements on all processors. For example, `real :: a(2,3)[*], b[*]` declares a 2×3 real *array coarray* and a real *scalar coarray*. Any processor can then read/write from/to a coarray on any image, including itself, using coarray *cosubscript*. A reference to a variable with no cosubscript is always a reference to the local variable. For example, line `b=a(1,1)[3]`, executed by any

image, will copy element `(1,1)` of array `a` from image 3 to local variable `b`; line `a(2,:)=b[1]`, executed by any image, will set all elements in row 2 of array `a` on that image to the value of `b` on image 1. The total number of images is typically set at run-time via environment variables and is available to the programmer via `num_images` intrinsic. The programmer can control which image executes what lines using `this_image` intrinsic.

While all local arrays declared as coarrays are independent, together they can be thought to represent one large global array. The interpretation of such imaginary global array is entirely up to the user. Thus in the coarray model, the global address space is not partitioned but rather *assembled*.

3 CA Coarray Data Structures

The major CA model data structure is the cellular array itself. We define it as a 4D coarray with a 3D image grid: `integer, allocatable :: space(:,:,:,:)[:,:,:]`. The 3D image grid is chosen because it minimises the amount of internal halo exchange information. The array has three spatial dimensions plus an extra dimension to store multiple physical quantities (layers). At present the library provides for two layers: (1) grains and (2) damage.

In addition to the main `space` coarray, all other data structures which need to be accessed by all images are defined as coarrays. Examples are: grain volume coarray: `integer, allocatable :: gv(:)[:,:,:]` and coarray of grain rotation tensors: `real, allocatable :: rt(:,:,:)[:,:,:]`. Note that it makes sense to allocate these coarrays with the same *codimensions*.

4 Halo Exchange

Each cell in the model represents a cube of material at some user-defined scale. We use a 26-cell nearest neighbourhood, i.e. $3 \times 3 \times 3$ cells minus the central cell. Part of the neighbourhood of a boundary cell resides in an array on another image, hence halo exchange between neighbouring images is necessary before every cell evolution (solidification, coarsening, fracture, etc.) increment. To store the halos, the `space` coarray is allocated with size increased by 2 cells in each direction. The halo exchange is done in parallel. Consider the following simple halo exchange code, running on `n` images, where the user interprets the `space` coarray as sharing a common face normal to direction 1.

```
allocate( space(0:11,0:11,0:11) [n,1,*] )
integer :: imgpos(3), lcob(3), ucob(3)
imgpos = this_image( space )              ! Image location in the coarray grid
lcob = lcobound( space )                  ! Lower cobound of space coarray
ucob = ucobound( space )                  ! Upper cobound of space coarray
if ( imgpos(1) .ne. lcob(1) ) &           ! For all images but the leftmost
 space(0, 1:10,1:10) =        &           ! copy the upper boundary cell states
 space(10,1:10,1:10)          &           ! into the lower halo array
  [imgpos(1)-1,imgpos(2),imgpos(3)]
if ( imgpos(1) .ne. ucob(1) ) &           ! For all images but the rightmost
 space(11,1:10,1:10) =        &           ! copy the lower boundary cell states
 space(1, 1:10,1:10)          &           ! into the upper halo array
  [imgpos(1)+1,imgpos(2),imgpos(3)]
```

In this example each image has 10^3 microstructure cells plus 6(faces) $\times$ 100 + 12(edges) $\times$ 10 + 8(corners) = 730 halo cells. The production code (`hxi`) is a lot longer because of the need to exchange halos is three directions plus edges and corners. The library also includes a global halo exchange routine, (`hxg`) called when self-similar boundary conditions on the whole model are chosen by the user.

5 I/O

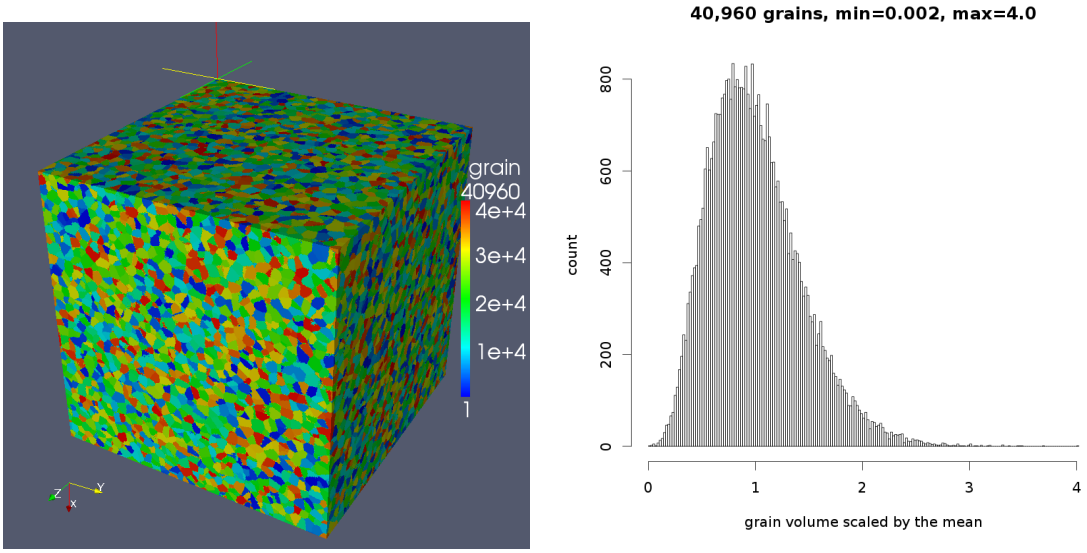
Coarray I/O can be done in at least two ways. Each image can write its own coarray array in a unique file. Then it is a job of the post-processing program to read and process all these files in the correct order. This approach is particularly attractive if the post-processing program supports parallel I/O. For example, Paraview (http://paraview.org) provides a parallel XDMF (http://xdmf.org) reader. This option has not be explored yet. Instead, we have implemented a serial routine that writes data from all images to a single binary file in correct order, as shown below. We then use the Paraview binary reader for visualisation. This is not very efficient, but simple. As long as the state of the model is written to file occasionally, this option seems acceptable. However, if the model has to be written to file often, then this serial routine will become a bottleneck.

```
lb = lbound(coarray) + 1                          ! Lower and upper bounds of
ub = ubound(coarray) - 1                          ! the coarray with no halos.
lcob = lcobound(coarray)                          ! Lower and upper cobounds
ucob = ucobound(coarray)                          !

if (this_image() .eq. 1) then                     ! Only img 1 does the writing

open(unit=iounit, file=fname, access="stream", & ! Open file for binary
 form = "unformatted", status = "replace")        ! stream write access

do coi3 = lcob(3), ucob(3)                        ! Nested loops
 do i3 = lb(3), ub(3)                             ! for writing
  do coi2 = lcob(2), ucob(2)                      ! in correct order
   do i2 = lb(2), ub(2)                           ! from all images.
    do coi1 = lcob(1), ucob(1)                    !
     write( unit = iounit )        &              ! Write one column at a time
      coarray(lb(1):ub(1),i2,i3)
       [coi1, coi2, coi3]
    end do
   end do
  end do
 end do
end do

end if
```

6 Model predictions

The library is distributed with a number of test programs, each calling different combinations of the library routines. The tests were done on HECToR phase 3, which is a Cray XE6 computer (www.hector.ac.uk). All tests were run with high resolution, 10^5 cells per grain, required to achieve scale independence [3].

Fig. 1(a) shows the predicted *equiaxed* microstructure typically found in normalised steels. This was obtained with `space(200,200,200)[8,8,8]` i.e. using 4.1×10^9 cells and 40,960 grains. On 512 processors (16 nodes, 32 processors per node) the run took 5m 36s wall time. The resulting output file was 16GB. Fig. 1(b) shows grain size (volume) histogram obtained from the solidification model shown in Fig. 1(a). This data is vital for model validation because it can be directly compared against experiments. Fig. 2(a) shows the histogram of the grain mis-orientation angle, for a material with no texture. The mis-orientation angle is an important parameter affecting grain boundary migration and fracture behaviour. The histogram agrees perfectly with the theoretical distribution. Finally Fig. 2(b) shows a cleavage crack propagating through a single, randomly oriented grain. The direction and the value of the maximum principal stress comes from the finite element solver. The importance of this result is in demonstrating the fracture predictive capability of the CA fracture model.

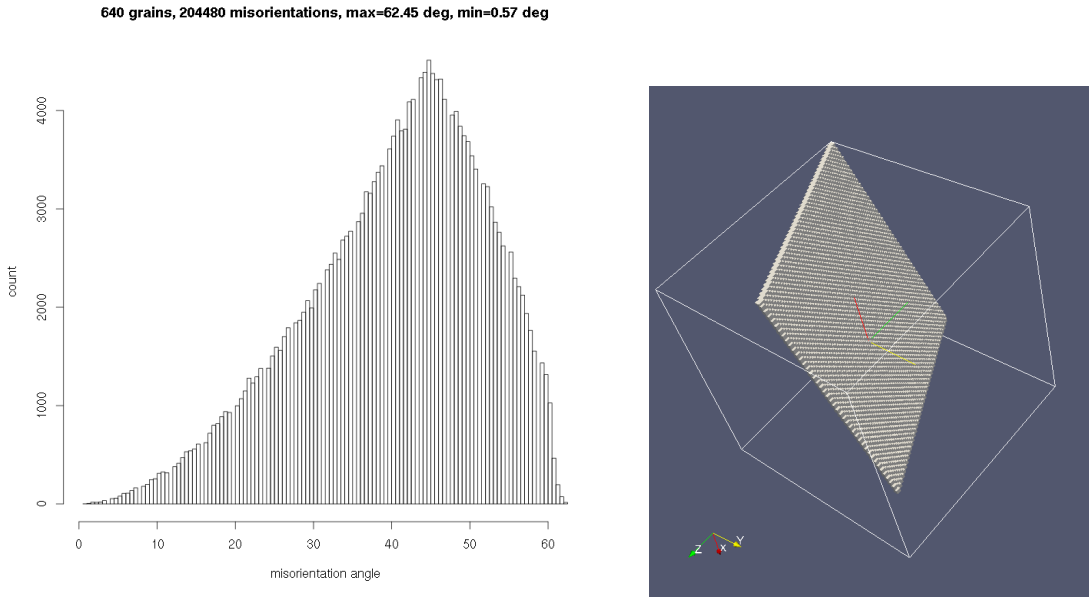


(a) (b)

Figure 1: A 4.1×10^9 cell, 40,960 grain equiaxed microstructure model, showing (a) grain arrangement with colour denoting orientation; (b) grain size size (volume) histogram.

7 Synchronisation, Performance and Code Optimisation

The Fortran standard provides a global barrier, `sync all`, and selective synchronisation between arbitrary image sets via `sync images`. Both types are used in this CA library. In addition, allocating or deallocating a coarray causes implicit synchronisation of all images.



(a) (b)

Figure 2: Examples of the CA coarray library use for microstructural modelling, showing (a) a grain mis-orientation histogram and (b) a cleavage crack in a randomly oriented single grain.

Synchronising a library is hard. This is because the order in which different routines can be called, if called at all, is not predictable. In addition, the user might write code where different images call different routines, potentially causing deadlocks.

One extreme option is to synchronise every routine on entry and exit, which, of course, is a performance killer, but removes the responsibility from the user to ever worry about synchronisation, provided they call the same routines in the same order from all images. Another extreme is not to provide any synchronisation in the library routines, but delegate all synchronisation to the user. This option is easy to implement, but is extremely error prone for the user.

We chose to synchronise only those routines where synchronisation is required for correct execution. For example, halo exchange routines do not include synchronisation statements because they are not required there. Images can do halo exchange in any order. However, for correct operation, these routines must be called by every image and only when no image is updating the cell states. This means that in practice some synchronisation must be used before and after a call to the halo exchange routine. This responsibility is left to the user. In contrast, the solidification routine uses synchronisation because it calls the halo exchange routine each iteration, and because it has to do a collective reduction operation, to determine when the model space has solidified on all images.

Using only minimal required synchronisation and avoiding single image computation are the key to a good performance.

As an example, here we analyse the performance of the solidification routine, that has a triple nested loop for all model cells in the image. Inside the loop, if a cell state is 'liquid', then it acquires the state of a randomly chosen neighbour. The loop cannot be replaced by a parallel `do concurrent` construct because it uses `random_number` intrinsic which is not `pure`, It might

be possible to parallelise the loop with OpenMP, however, we haven't explored this yet.

```
integer :: fin
logical :: finished

main: do
  array = space( :, :, :, type_grain )
  do x3 = lbr(3),ubr(3)
  do x2 = lbr(2),ubr(2)
  do x1 = lbr(1),ubr(1)
    if ( space( x1, x2, x3, type_grain ) .eq. liquid ) then
      call random_number( candidate )     ! 0 .le. candidate .lt. 1
      z = nint( candidate*2 - 1 )         ! step = [-1 0 1]
      array( x1,x2,x3 ) = space( x1+z(1), x2+z(2), x3+z(3), type_grain )
    end if
  end do
  end do
  end do
  space( :, :, :, type_grain ) = array

  finished = all( space( lbr(1):ubr(1), lbr(2):ubr(2), lbr(3):ubr(3), &
                  type_grain ) .ne. liquid )
  fin = 1
  if (finished) fin = 0

!!!  exit if (fin .eq. 0) on *all* images

end do main
```

A global reduction, over all images, is required, to calculate `sum(fin)` over all images. If it has a non-zero value, then the iterations continue and another run of the triple loop is done. If the global sum of `fin` is zero, then the model is fully solidified on all images, and the iterative process is complete.

Fig. 3 shows four reduction strategies, which we have used for the purpose of a speed-up analysis. In all cases `img = this_image()` and `nimgs = num_images()`. The easiest method, shown in Fig. 3(a), is where a single image is reading values from all other images and computes the sum. All other images wait. A single `sync all` global barrier is required in this case.

A slightly more complex strategy, shown in Fig. 3(b), is where every image adds its value to the global total, kept on image 1. This is done one image at a time with a pair of matching `sync images` statements. However, a global barrier, `sync all`, is still required to make all images wait for image 1, before they can read the updated value from it.

In both strategies (a) and (b) only one image at a time is doing the work, which is very inefficient. An improvement can be achieved with a *divide & conquer* type of algorithm (or a binary search), Fig. 3(c). This example algorithm works only when the number of images is a power of 2, i.e. `num_images()`$= 2^p$. The loop takes only p iterations. On the first loop iteration all even images add their values to lower odd images. On each following iteration the step between the images in each pair increases by a factor of 2. On the last iteration image `num_images()+1` adds its total value to that of image 1. For the case of $2^3 = 8$ images ($p = 3$) this can be schematically illustrated like this: $i = 1 : (1) \leftarrow (2), (3) \leftarrow (4), (5) \leftarrow (6), (7) \leftarrow (8)$;

$i = 2 : (1) \leftarrow (3), (5) \leftarrow (7)$; $i = 3 : (1) \leftarrow (5)$. Synchronisation is done with pairs of matching `sync images` statements. As in case (b), `sync all` is required at the end to make all images wait for image 1 before copying its total value.

The final algorithm uses `co_sum` collective, Fig. 3(d), which at this time is still a Cray compiler extension, but should become an intrinsic function when TS 18508 [7] is adopted into the Fortran standard. No synchronisation is required in this method. This is because using a collective subroutine in this case satisfies the two rules of the technical specification [7]. Rule 1: 'If it is [collective] invoked by one image, it shall be invoked by the same statement on all images of the current team in execution segments that are not ordered with respect to each other'. Rule 2: 'A call to a collective subroutine shall appear only in a context that allows an image control statement'. Hence, there is no possibility of any two images entering `co_sum` at different loop iterations. This ensures that all images exit the main loop at the same iteration count. Likewise, there is no possibility of a deadlock.

```
if ( img .eq. 1) then
  save_fin = fin
  do i = 1,nimgs
    fin = fin + fin[i]
  end do
  fin = fin - save_fin
end if

sync all
fin = fin[1]
```

(a) image 1 only

```
if ( img .ne. 1) then
  sync images ( img - 1 )
  fin[1] = fin[1] + fin
end if

if ( img .lt. nimgs ) &
 sync images ( img + 1 )

sync all
fin = fin[1]
```

(b) all images in turn with `sync images`

```
step    = 2
stepold = 1

redu: do i = 1,p
  if ( mod(img,step) - 1 .eq. 0 ) then
    sync images ( img + stepold )
    fin = fin + fin[ img + stepold ]
  else if ( mod(img+stepold,step) - 1 .eq. 0 ) then
    sync images ( img - stepold )
  end if
 stepold = step
 step = step * 2
end do redu

sync all
fin = fin[1]
```

(c) divide & conquer with `sync images`

```
call co_sum( fin )
```

(d) `co_sum`

Figure 3: Four reduction algorithms and associated synchronisation strategies used in the speed-up analysis.

Fig. 4 shows scaling of the solidification routine with the above four different implementations of the collective operations. These tests were run on a model with 2^{30} cells, using from $2^3 = 8$ to $2^{15} = 32768$ cores, i.e. with the model coarray defined from `(512,512,512)[2,2,2]` to `(32,32,32)[32,32,32]`. The times were calculated with `cpu_time` intrinsic. Each test was run 3 times, and the error bars show the fastest and the slowest times.

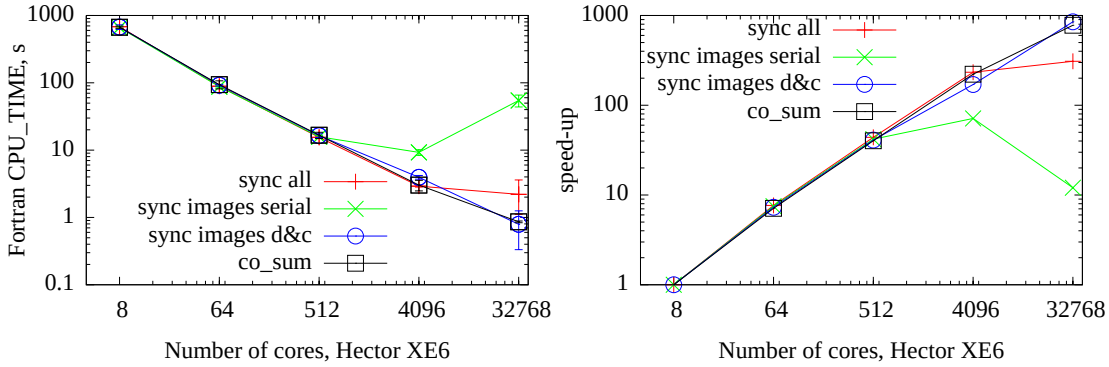


Figure 4: Timing and speed-up of the solidification routine with different implementations of the collective operation and corresponding synchronisation.

For up to 512 cores, there are no differences between the four cases. However, for higher core counts, the `co_sum` and the divide & conquer reductions show the best speed-up. For these two strategies the speed-up is nearly linear (on the log-log scale) up to at least 2^{15} cores. The speed-up is nearly 1000 for the core count raising by a factor of 2^{12}, from 2^3 to 2^{15}. We could not conduct experiments with $2^{16} = 65536$ cores due to budgetary limitations. Nevertheless, these speed-up results are very encouraging.

In addition to the scaling analysis, we have also performed profiling analysis of the solidification routines using CrayPAT API calls [9]. Profiling was done with a 2^{32} cell model run on 512 images. Tab. 1 shows the profiling results for reduction strategies (a), (c) and (d) in Fig. 3. All values, except the last row, are percentages of the total time spent in each part of the code. Note that model (d), with `co_sum` spends twice as long doing the global reduction as the triple loop computation. This is the exact opposite of strategies (a) and (c), calculation on image 1 with `sync all` and divide & conquer, where the triple loop takes roughly twice as long as the global reduction. Nevertheless, the `co_sum` approach is twice as fast overall.

	(a), `sync all`	(c), D&C	(d), `co_sum`
triple loop	50	60	25
global reduction	35	25	61
serial reduction + I/O	10	10	2
halo exchange	5	5	12
Time, s	35.0	47.1	19.7

Table 1: Relative times spent in different parts of the solidification routines and the total run time in seconds.

Another example of a global reduction operation is when calculating grain volumes: `integer, allocatable :: gv(:)[:,:,:]`. First each image calculates its local grain volume coarray

array. Then a global sum is calculated over all images. We note that using `co_sum` on 512 cores is 2 orders of magnitude faster than a single image computation.

8 Concluding Remarks and Future Work

The coarray CA microstructural library gives useful predictions of solidification, grain size distribution and cleavage fractures. Most of the code scales well up to 30k cores. A notable exception is the output routine, which is serial. The standard forbids a file to be connected to more than one image. This makes designing parallel I/O with coarrays hard. Intrinsic collectives, such as `co_sum`, lead to a better performance compared to user written collective routines. There are many triple nested loops in the library routines. It is possible that these can be optimised with OpenMP. Note that at present only Cray compiler supports OpenMP with coarrays. Finally, we are looking for a suitable scalable parallel open source finite element code to interface with the library, and ParaFEM (http://parafem.org.uk) will be our first choice.

9 Acknowledgments

This work made use of the facilities of HECToR, the UK's national high-performance computing service, which is provided by UoE HPCx Ltd at the University of Edinburgh, Cray Inc and NAG Ltd, and funded by the Office of Science and Technology through EPSRC's High End Computing Programme. This work also used the computational facilities of the Advanced Computing Research Centre, University of Bristol - http://www.bris.ac.uk/acrc/.

References

[1] A. Shterenlikht and I. C. Howard. The CAFE model of fracture – application to a TMCR steel. *Fatigue and Fracture of Engineering Materials and Structures*, 29:770–787, 2006.

[2] S. Das, A. Shterenlikht, I. C. Howard, and E. J. Palmiere. A general method for coupling microstructural response with structural performance. *Proceedings of the Royal Society A*, 462(2071):2085–2096, 2006.

[3] J. Phillips, A. Shterenlikht, and M. J. Pavier. Cellular automata modelling of nano-crystalline instability. In *Proceedings of the 20th UK Conference of the Association for Computational Mechanics in Engineering, 27-28 March 2012, the University of Manchester, UK*, 2012.

[4] G. Guillemot, Ch.-A. Gandin, and M. Bellet. Interaction between single grain solidification and macro segregation: Application of a cellular automaton - Finite element model. *Journal of Crystal Growth*, 303:58–68, 2007.

[5] D. Raabe and R. C. Becker. Coupling of a crystal plasticity finite-element model with a probabilistic cellular automaton for simulating primary static recrystallization in aluminium. *Modelling and Simulation in Materials Science and Engineering*, 8:445–462, 2000.

[6] ISO/IEC 1539-1:2010. *Fortran – Part 1: Base language, International Standard*. 2010. http://j3-fortran.org/doc/standing/links/007.pdf.

[7] ISO/IEC JTC1/SC22/WG5 N1983. *Additional Parallel Features in Fortran*. JUL-2013. ftp://ftp.nag.co.uk/sc22wg5/N1951-N2000/N1983.pdf.

[8] I. D. Chivers and J. Sleightholme. Compiler support for the fortran 2003 and 2008 standards, revision 12. *ACM Fortran Forum*, 32(1):8–19, 2013. http://www.fortranplus.co.uk/resources/fortran_2003_2008_compiler_support.pdf.

[9] Cray Inc. *Using Cray Performance Measurement and Analysis Tools*. MAR-2013. http://docs.cray.com/books/S-2376-610/.